

# A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures

Gemma Grau, Xavier Franch

Universitat Politècnica de Catalunya (UPC)  
c/ Jordi Girona 1-3, Barcelona E-08034, Spain.  
{ggrau, franch}@lsi.upc.edu

**Abstract.** There is a recognized gap between requirements and architectures. There is also evidence that architecture evaluation, when done at the early phases of the development lifecycle, is an effective way to ensure the quality attributes of the final system. As quality attributes may be satisfied at a different extent by different alternative architectural solutions, an exploration and evaluation of alternatives is often needed. In order to address this issue at the requirements level, we propose to model architectures using the *i\** framework, a goal-oriented modelling language that allows to represent the functional and non-functional requirements of an architecture using actors and dependencies instead of components and connectors. Once the architectures are modelled, we propose guidelines for the generation of alternative architectures based upon existing architectural patterns, and for the definition of structural metrics for the evaluation of the resulting alternative models. The applicability of the approach is shown with the Home Service Robot case study.

## 1 Introduction

There is a gap between requirements and architectures which is mainly due to the fact they use different terms and concepts to capture the model elements relevant to each one [20]. However, there is a connection between architectural design decisions and the quality attributes of the final system and, so, it is possible to analyse and to evaluate an architecture in the context of the goals and requirements that are levied to the systems that will be build from it [8]. On the other hand, quality attributes may be satisfied by different alternative architectural solutions (i.e., architectural patterns) and, so, there is often the need of exploring and evaluating several alternatives.

In this paper we propose to address the generation and evaluation of architectures at the requirements level by using a goal-oriented approach. Goal-oriented models allow expressing the intentional concepts using the same constructs for the requirements and for the architectures. We consider that they are an adequate formalism for representing software architectures because they allow expressing usual architecture-related concepts such as component, node, file, resource, dependency and so on. Additionally, goal-oriented models are becoming intensively used in fields such as requirements engineering and organizational process modelling, which has two main implications. In the one hand, transition from organizational and system

models to architecture models can be smoother due to the use of the same formalism, and even traceability benefits from this. On the other hand, contributions and findings made in these other fields can be assessed and eventually incorporated into software architecture modelling and analysis.

Because of that, goal-oriented models [25], [28] have already been used for representing software architectures addressing the gap between requirements and architectures [25]. Among the different existing goal-oriented proposals, we remark the *i\** framework [27], a goal-oriented modelling language that allows representing the functional and non-functional requirements of an architecture using actors and dependencies instead of components and connectors. For more details on the adequacy of using *i\** models for representing and analyzing software architectures, we refer to [17].

In this paper we use the *i\** framework in order to support the generation and evaluation of alternative architectures. We are interested in doing this process in a reliable way and, thus, we propose a set of guidelines for performing both activities in a systematic manner. Therefore, the generation of alternatives is based on the use of architectural patterns [6] and the evaluation of alternatives is done by applying structural metrics [14] over the produced models. In order to show the applicability of our approach, we apply the proposed steps and guidelines to the Home Service Robot case study as presented in [22], [23].

The remainder of this paper is organized as follows. The problem statement for the Home Service Robot case study is presented in section 2. In section 3 we present a brief introduction to the *i\** framework. In our approach we do not consider the generation and evaluation of alternative architectures as an isolated process and, so, the context of applicability and tool support are presented in section 4. In section 5 we propose a set of guidelines for the generation of alternative architectures based on existing architectural patterns. Our proposal of metrics for evaluating the resulting models and a set of guidelines for defining the metrics are presented in section 6. Finally, section 7 presents the conclusions and future work.

## 2 The Home Service Robot Case Study

This case study is based on the problem statement of the prototype of a Home Service Robot (HSR) for daily services provided in [22], [23]. The case study has been chosen because the details about the problem statement are very clear, it is simple enough to be analysed and understood in the context of a paper, and also because mobile robots are commonly used in software architecture examples [26].

According to [22], [23], the HSR is a prototype that supports the following daily home services:

- **Call and Come (CC).** This service analyzes the audio data sampled in order to detect predefined sound patterns. If a “come” command is recognized, the robot tries to detect the direction of the sound source, rotates to the direction of the sound source and tries to recognize a human. If the caller’s face is detected, the robot moves forward until it reaches within 1 meter from the caller. If a “Stop” command is recognized while the robot is moving, the robot stops.

- **User Following (UF).** The robot locates a user and constantly checks vision data and sensor data for keep following the user. The robot follows the user within 1 meter range. If the robot misses the user, it notifies him by saying “I lost you” and the action terminates.
- **Security Monitoring (SM).** The robot patrols around a house for surveillance using a map. Intrusion or accidents are defined as patterns recognizable from vision and sound data. If such an event is detected, the robot notifies the user directly via an alarm or indirectly through a home server.
- **Tele-presence (TP).** A remote user can control the robot using a PDA. The robot sends the remote user a map of the house and the user can command the robot to move to a specific position. In addition, the robot can send captured images to the remote PDA for surveillance.

In order to provide those services, the HSR has the following hardware components: a Single Board Computer that controls the peripherals; a Front Camera to allow face recognition, user tracking, security monitoring and tele-presence; a Ceiling Camera to do map building and self-positioning; 8 SL Microphones to interpret speaker commands and locate its specific position; a Structured Light Sensor to detect obstacles and recognize footsteps; an Actuator to allow the HSR movement; an LCD display to show information; a Wireless Lan to communicate to the Home Server; and, finally, a Speaker to generate sound.

For more details about the HSR problem statement we refer to [22], [23].

### 3 The $i^*$ Framework

The  $i^*$  framework proposes the use of two types of models for modelling systems, each one corresponding to a different level of abstraction: the Strategic Dependency (SD) model represents the intentionality of the process and the Strategic Rationale (SR) model represents the rationale behind it. SD models focus on the relationships among different actors that cooperate for satisfying some goals and, so, they are more interesting from the architecture point of view. Consequently, in this paper we focus on SD models.

A SD model consists of a set of nodes that represent actors and a set of dependencies that represent the relationships among them. Dependencies express that an actor (*dependor*) depends on some other (*dependee*) in order to obtain some objective (*dependum*). Thus, the *dependor* depends on the *dependee* to bring about a certain state in the world (goal dependency), to attain a goal in a particular way (task dependency), for the availability of a physical or informational entity (resource dependency) or to meet some non-functional requirement (softgoal dependency). These four types of *dependum* allow two different types of relationships: *intentional*, representing what behaviour a component expects from other parts of the system; and *operational*, representing how one component communicates with other parts of the system. Therefore, intentional relationships are represented by:

- **Goal dependencies** stating functional requirements, e.g. the User depends on the HSR for the goals *Come when called* and *Accidents are avoided*.
- **Softgoal dependencies** stating high-level non-functional requirements, e.g. the

User depends on the HSR depends for the softgoals *User position location is accurate* and *Robot movement is efficient*.

- **Resource dependencies** stating flow of concepts, and remarkable some type of knowledge, or a concept, relevant for the domain that does not physically exist, e.g. the concept *Voice Command* in the context of the HSR.

On the other hand,  $i^*$  may be also used to represent architectural concerns by means of the following operational relationships:

- **Task dependencies** stating service invocation, e.g. the HSR depends on the User for the tasks *Introduce position in map* and *Introduce Sound patterns*.
- **Resource dependencies** stating information interchange, e.g. the HSR depends on the user for the resource *Tele-surveillance images*.
- **Goal dependencies** stating fit criteria for non-functional requirements, e.g. the HSR *Ensures security by avoiding obstacles*. Note thus that non-functional requirements change from being represented as soft goals to goals.

For more details about  $i^*$ , we refer to [21], [27]. The graphical notation is shown in Fig. 1. using, as an example, the SD dependencies between the HSR and the User.

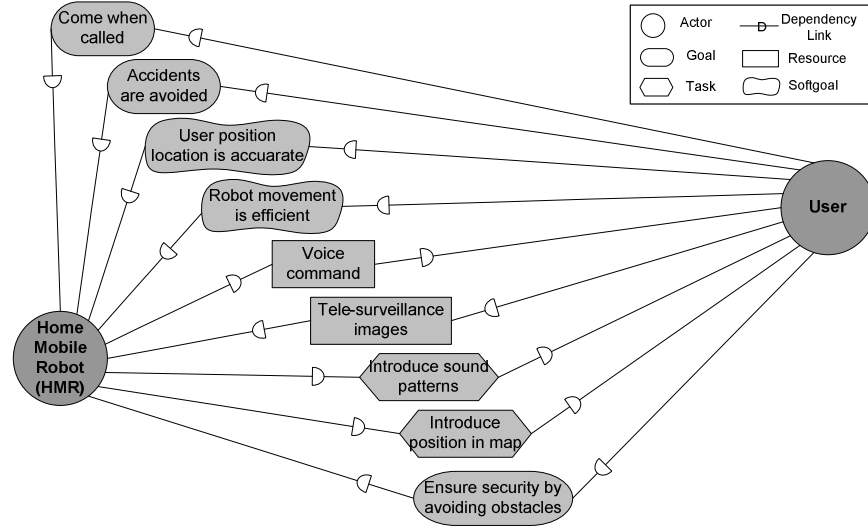


Fig. 1. Excerpt of an  $i^*$  model for the HSR case study

## 4 Context of Applicability and Tool Support

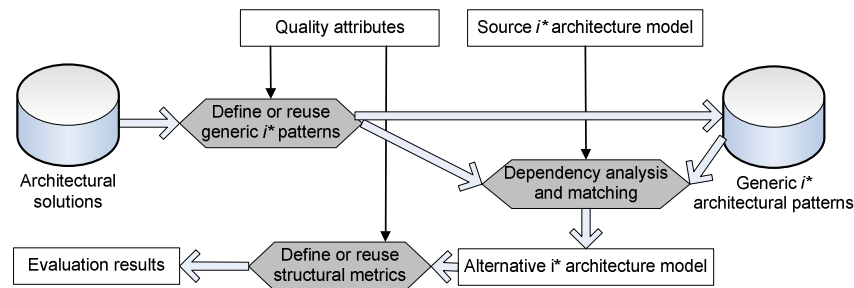
In our approach we do not consider the generation and evaluation of alternative architectures as an isolated process but as a part of a reengineering framework, that we have named Reef [16]. The reasons for such a claim are twofold. On the one hand, most of the reengineering approaches consider the generation and evaluation of alternative solutions. On the other hand, in most of the cases, it is possible to state the

premise that there is always a current process undertaken by humans or by a legacy system that can be used as a departing point for reengineering. Several proposals in the field of software architectures also follow a reengineering approach, among them we remark [3], [22], [23].

We have refined the Reef generic framework into SARiM [16], a Software Architecture Reengineering *i\** Method, which is composed of the following phases: 1) Analysis of the source software architecture; 2) Conceptualization of the analysed software architecture into an *i\** model; 3) Elicitation of new requirements for the software architecture; 4) Exploration of candidate software architecture solutions; 5) Assessment of the generated solutions using evaluation techniques; and 6) Creation of the specification for the new software architecture. In this paper we focus on phases 4 and 5. However, we assume that other existing techniques have been used for the first three phases and, so, before the generation and evaluation of architectures, there is an existing source *i\** architecture model and a set of quality attributes to be used as the starting point. These artefacts can be generated with many existing techniques, among which, we have chosen the PRiM method [19] for generating the source *i\** architecture model, and KAOS [9] for obtaining the requirements and quality attributes.

In Fig. 2 we provide an overview of the proposed process. We can observe that the generation of alternatives begins with the definition of the generic  $i^*$  architectural patterns. These patterns are constructed using already existing architectural solutions, which are selected according to the quality attributes to be achieved. Once the generic  $i^*$  architectural patterns are defined, the alternative architectures are generated by applying a dependency analysis and matching process over the source  $i^*$  architecture model and each selected generic  $i^*$  architectural pattern, resulting to set of alternative  $i^*$  architecture models. We remark that the generic  $i^*$  architectural patterns can be stored and reused when reapplying the process. Finally, once the alternative  $i^*$  architecture model are generated, they can be evaluated by defining or reusing structural metrics, providing the evaluation results that would assess the selection of the most suitable architecture.

In order to perform the generation and evaluation of alternatives in a reliable way, tool support is needed. We propose to use J-PRiM [18], a tool that supports the first five phases of Reef using the techniques proposed in PRiM [19] and, thus, it has been adapted to the generation and evaluation of alternatives that we propose.



**Fig. 2.** Overview of the process for the generation and evaluation of alternative architectures

## 5 Generation of the Alternative Architectures

There are several existing proposals on the generation of architectures, for instance [3], [4], [20], among others. These methods have in common that they all use an existing architecture specification as a departing point; they all propose the use of well-known architectural solutions for the generation of alternatives; and, they all choose the solutions according to previously obtained quality attributes.

On the other hand, existing work on the generation of alternative architectures using the  $i^*$  framework as a modelling language adopt a similar approach. This work is mainly represented by [24], [2], and it is oriented towards agent-oriented software architectures. In their context, patterns are used for generating organizational architectures which are represented in  $i^*$ . Based on these patterns, the  $i^*$  models are build by matching the concepts represented in the  $i^*$  organizational patterns with the functional and non-functional requirements for the new software architecture.

Based on this existing work, our approach for the generation of the alternative software architectures proposes four guidelines that transform existing architectural solutions into generic  $i^*$  architectural patterns and, then, use a matching process between the dependencies expressed in the source  $i^*$  architecture model and the ones defined in the generic  $i^*$  architectural patterns. These guidelines can be intertwined and iterated as needed. We remark that guidelines 1.1 and 1.2 are only applied once for each architectural pattern, allowing reusability when reapplying the method.

- **Preliminaries: Pattern Selection.** There are many architectural patterns that can be used for generating software architectures. So, in order to generate alternative architectures in a controlled way, we will only explore those patterns that help the achievement of the quality attributes we want for the final software system. The way these quality attributes are elicited from the stakeholders remains out of the scope of this paper, however, we mention that most of the quality attributes will be represented as softgoals in the  $i^*$  model. A way to select the patterns is to use the NFR [7] approach for modelling the contributions of each softgoal to the architectural patterns and, then, select those with a positive contribution. It is also possible to check the properties of each pattern in a pattern catalogue [6], a feature-solution graph [4], or a Property to Style Mapping Table [20], among others.

For instance, if the quality attribute to achieve is Maintainability we may select a Blackboard pattern, and if Exchangeability is needed, we may select a layered architecture. None of these architectural solutions have to be selected if efficiency is a crucial point for the architecture.

- **Guideline 1.1: Actor Identification.** Once the pattern is selected, we analyse it in order to identify the architectural components suggested by the pattern. Each component will be modelled as an actor in the  $i^*$  model of the new alternative architecture.

For instance, in the *Blackboard architectural pattern* as defined in [6], three actors are identified: the Blackboard, the Knowledge Source and the Control. We remark that, according to the pattern documentation, several Knowledge Sources can be used. Moreover, in some cases, the specific number and name of the components remains undefined in the pattern. For instance, that's the case of the *Layers architectural*

*pattern* as defined in [6]. In this situation, in order to discover all the actors we have first to determine the number of layers and the abstraction level that they represent. This can be done by applying our own criteria or by adapting the criteria used to define other layer architectures, such as the OSI 7-Layer Model or the TCP/IP protocol [6].

- **Guideline 1.2: Definition of the generic  $i^*$  architectural pattern.** Once the actors are defined, the architectural solution is deeper analysed in order to abstract the general responsibilities of each actor and the generic dependencies with the other actors. As a result we obtain a generic  $i^*$  architectural pattern. The information needed for such an analysis is the one documented in the architectural solution. In order to enforce the link between requirements and architectures when deciding the kind of a certain dependency, we propose to adapt the six CBSP architectural dimensions proposed in [20] into the  $i^*$  framework:

- *Task dependencies* model those elements that describe or involve processing components.
- *Resource dependencies* model those elements that describe or involve data components.
- *Goal dependencies* model those elements that that describe system-wide features or features pertinent to a large subset of the system's components or connectors.
- *Softgoal dependencies* model those elements that describe or imply data or processing component properties, bus properties or system properties.

In order to allow further reuse of the documented generic  $i^*$  architectural pattern the source of the pattern and the decisions taken during its definition have to be documented.

In Fig. 3. we show how we have defined the generic  $i^*$  architectural pattern for the *Blackboard architectural pattern*. At the left of the figure we can see the classes and their responsibilities as they are documented in [6]. We can also observe that we have added an  $i^*$  actor for each of the classes of the pattern. The dependencies have been established as follows:

- The Blackboard manages central data, which is a system feature and so, it is modelled as the goal dependency *Central data is managed*. As central data is a data component, a resource dependency *Central data* is also stated. As both the Knowledge Source and the Control depend on the Blackboard for the central data management, each dependency appears twice.
- The Control monitors the Blackboard and schedules the Knowledge Sources activations. Both are system features and so they are represented as goal dependencies. Thus, the Blackboard depends on the Control for *Blackboard is monitored* whilst the Knowledge Source depends on the Control for the goal *Knowledge source activations are scheduled*. We remark that the Control monitors the Blackboard by analysing the Central data (which is an already existing dependency). On the other hand, as the Control involves a process for scheduling the Knowledge Sources, we need a task dependency stating that the Control depends on the Blackboard for *Activate knowledge sources*.
- The Knowledge Source evaluates its own applicability by using the central data. Thus, the Blackboard depends on the Knowledge Source for the goal

*Knowledge source applicability is evaluated.* The Knowledge Source has the responsibility to compute a result (which involves a data component) and to update the Blackboard (which involves a processing component). Thus, the Blackboard depends on the Knowledge Source for the resource *Computed result*, and the Knowledge Source depends on the Blackboard for the task *Update blackboard*.

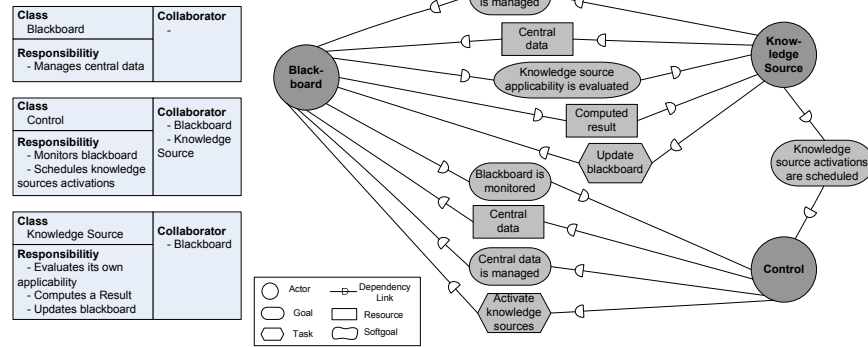


Fig. 3. Abstraction of the generic  $i^*$  pattern for the Blackboard architectural pattern

- **Guideline 1.3: Actors analysis and matching.** Using the source  $i^*$  architecture model and the generic  $i^*$  architectural pattern of the solution to be applied, we analyse the dependencies in both models in order to match the related elements and establish the equivalence between the source  $i^*$  architecture model actors and the generic  $i^*$  architectural pattern actors. As it is proposed in [17], in both groups we distinguish four kinds of actors:

- Human actors. i.e., the final users of the software system.
- Organizational actors. i.e. the organizations that provide or require services from the software system and its final users.
- Software actors. i.e., the software system that is in charge to satisfy the human actor requirements. The software system can be represented by a unique software actor or by a set of actors that represents components and interact one with each other.
- Hardware actors. i.e., the hardware devices in those software systems where we need to obtain certain information from the environment.

We remark that there are some actors on the source  $i^*$  architecture model that may not have an equivalence in the generic  $i^*$  architectural pattern and viceversa, for instance the actors that represent humans, organizational or hardware components in the source  $i^*$  architecture model are not typically actors of the generic  $i^*$  architectural patterns. This aspect is solved in the next guideline with the reallocation of responsibilities.

If we match the concepts of the Home Service Robot (HSR) and the *Blackboard architectural pattern* we can observe that the HSR involves a human actor (the User), a software actor (the Single Board Computer that is the component that controls the



HSR), and several hardware actors that interact with the user (i.e., Front Camera, Microphones, Actuator, etc., see section 2 for more details). However, we can also observe that, although the actors on the generic  $i^*$  architectural pattern and the ones on the source  $i^*$  architecture model conform two disjoint groups, the HSR software actor that controls the HSR can be refined into the set of actors proposed by Blackboard  $i^*$  architectural pattern.

The *Blackboard architectural pattern* contemplates the possibility of having several Knowledge Sources. The strategy we follow to decide the number of Knowledge Sources and their specific responsibilities is to analyse other existing blackboard configurations specific for robots. Among them we have chosen the one proposed in [26], which suggest the following Knowledge Sources, that we model as actors in the alternative  $i^*$  architecture model: the Lookout, which monitors the environment for landmarks; the Pilot, which is in charge that planning the current path and control the robot actuators; and, finally, the Map Navigator, which plans the high-level path. The actor Control of the Blackboard  $i^*$  architectural pattern corresponds to the Captain component in [26] and the hardware actors can be considered as the perception subsystem in [26].

- **Guideline 1.4: Reallocation of responsibilities.** Once the actors of the source  $i^*$  architecture model and the generic  $i^*$  architectural pattern have been analysed, we create the new alternative  $i^*$  architecture model with the following actors:
  - The software actors of the generic  $i^*$  architectural pattern.
  - The human, organizational, and hardware actors of the source  $i^*$  architecture model.

As the actors of the source  $i^*$  architecture model may not be considered on the generic  $i^*$  architectural pattern, the dependencies related with these actors have to be reallocated on the actors suggested by the pattern. This reallocation is done by matching the different elements in both models until having the entire source  $i^*$  architecture model dependencies represented following the structure of the generic  $i^*$  architectural pattern.

As a result of the matching activities we create a new alternative  $i^*$  architecture model with the human and hardware actors of the Home Mobile Robot source  $i^*$  architecture model and the software actors of the Blackboard  $i^*$  architectural pattern as it has been customized applying the previous guideline. Once this is done, the reallocation of responsibilities is carried out as follows: the processes for locating the user, analysing the distance with objects and detecting predefined intrusions patterns fall into the Lookout actor; the current path planning, including the control of the movement actuators (rotation and advance functions) fall into the Pilot actor; the analysis of the current position and the planning of the tele-surveillance path, remain inside the Map Navigator; and, finally, the interpretation of user commands and the monitoring of all his/her actions is done by the Control actor. Dependencies steaming from or going to the hardware actors remains unchanged on the hardware actors' side and are reallocated into the Blackboard actor in the Software side, the rest are reallocated in the software actors as mentioned.

## 6 Evaluation of the Alternative Architectures

There are many proposals that address the evaluation of alternatives, and there is also already existing work to compare the different evaluation techniques [10]. According to [8], there are several categories of evaluation techniques:

- *Questioning techniques* allow investigating any area of the project at any state of readiness and include scenario-based methods [3], [4];
- *Measuring techniques* require the existence of some artefact to measure and include the definition of metrics for an static analysis of the structure, being common to use an Architecture Description Language for that purpose; and,
- *Hybrid techniques* that combine elements from questioning and measuring techniques, such as the ATAM method [8].

In a deeper analysis of the techniques used in each category, we can observe that most of the methods that evaluate architectures at their early stages use scenario-based techniques, and that Architecture Description Languages represent a much lower level of detail and focus on the evaluation of the behaviour and performance.

There is also work that addresses the evaluation of alternatives modelled within the  $i^*$  framework. Despite that most of this proposals use reasoning-based techniques [27], structural metrics are also being used [5], [12], [13], [14].

Based on the structure of the  $i^*$  SD models, it is possible to analyse the degree of fulfilment of the quality attributes for each alternative architecture, which allows evaluating the generated alternatives and informing their selection. The quality attributes can be evaluated with metrics in the form proposed in [14]. Metrics are defined in terms of the actors (actor-based metrics) and the dependencies (dependency-based metrics) of the model. It is also possible to distinguish between global and local metrics, where global metrics give an overall value of the quality-attribute under consideration and local metrics uses maximum and minimum values to locate specific elements. As we want to evaluate generated architectures, we will only work with global metrics, for the definition and use of local metrics see [14].

- **Global actor-based metrics.** Given an architectural property  $P$  and an  $i^*$  SD model that represents a system model  $M = (A, D)$ , where  $A$  are the actors and  $D$  the dependencies among them, an actor-based architectural metric for  $P$  over  $M$  is of the form:

$$P(M) = \frac{\sum_{a: a \in A: \text{filter}_M(a) \times \text{correctionFactor}_M(a)}{\|A\|}$$

being  $\text{filter}_M: A \rightarrow [0,1]$  a function that assigns a weight to the every actor (e.g., if the actor is human, software or from a specific kind), and  $\text{correctionFactor}_M: A \rightarrow [0,1]$  a function that corrects the weight of an actor considering the dependencies stemming from or going to it.

- **Global dependency-based metrics:** Given an architectural property  $P$  and an  $i^*$  SD model that represents a system model  $M = (A, D)$ , where  $A$  are the actors and  $D$  the dependencies among them, a dependency-based architectural metric for  $P$  over  $M$  is of the form:

$$P(M) = \frac{\sum d: d(a,b,x) \in D: \text{filter}_M(d) \times \text{correctionFactor}_{M,dee}(a) \times \text{correctionFactor}_{M,der}(b)}{\|D\|}$$

being  $\text{filter}_M: D \rightarrow [0,1]$  a function that assigns a weight to the every *dependum* (e.g., if the *dependum* is goal, resource, task, softgoal if it is from a specific kind), and  $\text{correctionFactor}_{M,der}: A \rightarrow [0,1]$  and  $\text{correctionFactor}_{M,dee}: A \rightarrow [0,1]$  two functions that correct the weight accordingly to the kind of actor that the *dependor* and the *dependee* are, respectively.

In order to guide the definition of the filters and correction factors proposed by the metrics and perform the evaluation of the generated architectures, we propose the following guidelines.

- **Preliminaries: Quality Attributes Selection.** Quality attributes tend to be non-functional requirements or constraints that have already arisen in the previous phases of the method and, as such, they are modelled as softgoals in the source  $i^*$  architecture model. However, not all the quality-attributes are equally important and, thus, we have to choose the most relevant to the new architecture. This can be done using different techniques being one of them prioritising the requirements (e.g., by considering individual stakeholder ranking of properties).
- **Guideline 2.1: Defining the Evaluation Goal.** The Goal Question Metric (GQM) paradigm [1] is commonly used for defining metrics. For instance, in [11] the GQM is used to analyse what has to be measured. In our case, the scope of measurement is restricted, as we already know that we want to measure the degree on what the software architecture ensures a quality attribute. Thus, the general form of the evaluation goal will be:
  - To evaluate the <quality attribute> of the modelled software architecture **in order to assess it.**

For instance, the evaluation goal for assessing the quality attribute maintainability is:

  - To evaluate the maintainability of the modelled software architecture **in order to assess it.**
- **Guideline 2.2: Defining the Goal Questions.** Once the goal is defined, questions for evaluating the goal have to be defined, in the same way as it is proposed in [1] and applied in [11].
 

For instance, for assessing the goal defined for maintainability, the question is:

  - What elements do affect maintainability?

In the literature, there is evidence that maintainability is better achieved in those architectures that present a low level of coupling and a high level of cohesion [6].
- **Guideline 2.3: Defining the Goal Questions Metrics.** Metrics are used to assess the questions and, as we have explained at the beginning of this section, they can be actor-based or dependency-based according to [14]. For deciding the kind of metric we propose to define the following questions:

- What are the architectural elements that are more relevant for the quality attribute?

If the components are more relevant, we define an actor-based metrics. If the connections are more relevant we will define a dependency-based metric. Once the kind of metrics is defined, we have to choose the values to be assigned to  $\text{filter}_M(a)$  and  $\text{correctionFactor}_M(a)$  in actor-based metrics, and the ones for  $\text{filter}_M(d)$ ,  $\text{correctionFactor}_{M,der}(a)$  and  $\text{correctionFactor}_{M,dee}(a)$  in dependency-based metrics.

For guiding the selection of the most suitable structural element, we propose the set of questions shown in Table 1. The contents of the table was defined after a deep analysis of the structural elements on the  $i^*$  framework. This kind of analysis is similar to the one performed when applying metrics over UML Class Diagrams [15].

As a result, in Table 1 we present the set of questions for actor-based metrics. We observe that a certain actor can be filtered according to its kind and the specific component it represents. A correction factor can be applied if the number of dependencies related with the actor ( $\#dep(a)$ ) negatively affects the quality attribute; if only the dependencies where the actor is a depender ( $\#Dep_{er}(a)$ ) negatively affects the quality attribute; if only the dependencies where the actor is a depender ( $\#Dep_{ee}(a)$ ) negatively affects the quality attribute; or, if it is the total amount of actors related with the actor ( $\#actor(a)$ ) that negatively affects the quality attribute.

**Table 1.** Questions, answers and examples for stating the filters and correction factors of actor-based metrics

Metric element	Question	Answer	Example Value
1.1. Actor-based: $\text{filter}_M(a)$			
	Does the kind of the actor or the actor itself affects the quality attribute?		
		No	$\text{Filter}_M(a) = 1$
		Yes, the kind of component affects the quality attribute.	$\text{Filter}_M(a) = \begin{cases} w, & \text{if } a \in \text{Human} \\ x, & \text{if } a \in \text{Software} \\ y, & \text{if } a \in \text{Hardware} \\ z, & \text{otherwise} \end{cases}$
		Yes, the specific component affects the quality attribute	$\text{Filter}_M(a) = \begin{cases} m, & \text{if } a = \text{ActorA} \\ n, & \text{if } a = \text{ActorB} \\ \dots \end{cases}$
1.2. Actor-based: $\text{correctionFactor}_M(a)$			
	Does the actor dependencies or the actors related with the dependencies affects the quality attribute?		
		No	$\text{CorrectionFactor}_M(a) = 1$
		Yes, the number of dependencies affects it.	$\text{CorrectionFactor}_M(a) = \frac{1}{\#Dep(a)}$
		Yes, the number of dependencies ER affects it.	$\text{CorrectionFactor}_M(a) = \frac{1}{\#Dep_{er}(a)}$
		Yes, the number of dependencies EE affects it.	$\text{CorrectionFactor}_M(a) = \frac{1}{\#Dep_{ee}(a)}$
		Yes, the number of actors related with a affects it.	$\text{CorrectionFactor}_M(a) = \frac{1}{\#Actor(a)}$

We remark that both the filters and the correction factors can be further refined as needed until getting the desired level of detail. For instance, we may only be interested in the number of actors related with the actor that are of a certain kind or that represent and specific component. Also other arithmetical combinations are

possible, if they allow providing more accuracy in the results. Dependency-based metrics would be defined following a similar approach.

As we have mentioned before, maintainability is better achieved in those architectures that present a low level of coupling and a high level of cohesion. In the structure of the  $i^*$  models a low level of coupling can be measured by stating an actor-based metric, where the number of actors related with the current actor negatively affects the property:

$$\text{Actor-based coupling metric: } \text{filter}_M(a) = 1 \quad \text{and} \quad \text{correctionFactor}_M(a) = \frac{1}{\#Actor(a)}$$

In a similar manner, cohesion is related with the number of dependencies that steam from or goes through each actor. If the same dependency appears more than once, cohesion is damaged. In this case we can define a dependency-based metric as follows:

$$\text{Dependency-based cohesion metric: } \text{filter}_M(d) = \frac{1}{\#Duplicated(d)} \quad \text{and} \quad \begin{aligned} \text{correctionFactor}_{M,der}(d) &= 1 \\ \text{correctionFactor}_{M,dee}(d) &= 1 \end{aligned}$$

- **Guideline 2.4: Evaluating the Metrics.** The evaluation of the metrics is done by applying the corresponding actor-based or dependency-based formula with the values stated in the previous guideline. As alternative  $i^*$  architecture models can be large and complex, tool support is essential. As we have mentioned in section 4, we use J-PRiM [18] to support the evaluation of the alternatives according to the defined metrics.

In order to show the application of the metrics, we have generated and evaluated 4 different alternatives architectures for the HSR in J-PRiM [18]. In Fig. 3. we show an schema of how the dependencies are distributed according to the patterns: A) Blackboard; B) 8-Layers defining the 8 levels as proposed in [26]; C) 3-Layers defining the 3 levels as proposed in [6], and D) a Control-loop as defined in [26].

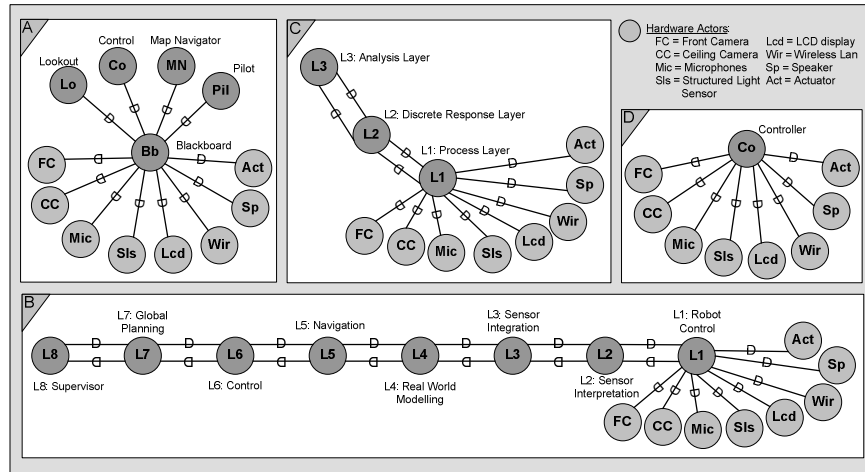


Fig. 3. Schema of the generated alternative  $i^*$  architecture models

The results of the evaluation are presented in Table 2. According to the coupling metric, we observe that those alternative  $i^*$  architecture models where there are more components and these components have dependencies with few other ones, score better for coupling (e.g., Layered architectures, being 8 levels better than 3). On the other hand, those alternative  $i^*$  architecture models where there are less dependencies for data interchange between different components, score better for cohesion (e.g., the Control loop architecture is more cohesive than the Layered architectures). Therefore, the solution that provides a better trade-off of this aspects is the Blackboard pattern.

**Table 2.** Evaluation results for the metrics indicating cohesion and coupling over 4 different architectural styles.

Property	Blackboard pattern	8-Levels layered architecture	3-Levels layered architecture	Control-loop architecture
Coupling	0.6250	0.5814	0.6065	0.8125
Cohesion	0.5217	0.1611	0.4000	0.95

## 7 Conclusions and Future Work

In this paper we present a set of guidelines for the generation and evaluation of alternative architectures. Our proposal uses the  $i^*$  framework, a goal-oriented modelling language that represents the software architecture functional and non-functional requirements using actors and dependencies between them. The guidelines assume that an initial source  $i^*$  architecture model and a set of relevant quality attributes have been obtained previously to the execution of the guidelines. From this point of view, we address the generation and evaluation of alternatives by adapting existing architecture solutions to the  $i^*$  framework by generating generic  $i^*$  architectural patterns. The elements on those patterns are analysed and matched against the ones on the source  $i^*$  architecture model in order to obtain the alternative  $i^*$  architecture models. Finally, these models are evaluated by applying structural metrics, which are defined by following a set of guidelines that follows the Goal Question Metric paradigm [1]. This process is supported by J-PR/M [18].

Among the benefits of the proposed approach we remark the following three. First, architectures are modelled at the early stages of the requirements process using a goal-oriented language, which we believe reduces the gap that is usually found between requirements and architectures. Second, it allows applying structural metrics directly on the requirements model, allowing the evaluation of alternative architectures without having to build any other artefact. Finally, as we are representing architecture-related concepts at the requirements level, we can benefit from the contributions and experience on both the use of the  $i^*$  framework and the research on the generation and evaluation of alternatives.

Regarding the capabilities to deal with the modelling, generation and evaluation of software architectures, our process satisfies the desiderata proposed in [26] as follows:

- **Composition.** The  $i^*$  framework allows describing a system as a composition of independent components and connections, where the components are represented

by actors and the connections are represented by means of dependencies between these actors.

- **Abstraction.** The  $i^*$  framework allows describing the components and their interaction at different abstraction levels. Thus, the system can be represented as a unique software actor or as a set of software actors representing the components of the software architecture.
- **Reusability.** Reusability is achieved at two levels. On the one hand, generic  $i^*$  architectural patterns are created only once for each architectural solution and can be used in other applications of the process. On the other hand, generated  $i^*$  architectures can be used as the source  $i^*$  architecture model in further iterations of the process.
- **Configuration.** The generated  $i^*$  architectures are based on existing architectural solutions, which clearly states that the system structure is independent from the elements being structured.
- **Heterogeneity.** It is possible to combine several architectural descriptions modelled within the  $i^*$  framework, and also to switch the level of detail they represent (for instance, from the whole system to the representation of architectural patterns or architectural styles).
- **Analysis.** We propose to analyse the resulting  $i^*$  models using structural metrics as proposed in [14], however other analysis techniques within the  $i^*$  framework can be used. They can be based on the structural properties of the  $i^*$  framework [2], [12], [13], or based on the reasoning capabilities it provides [27], [28].

As future work, we aim at creating a catalogue of generic  $i^*$  architectural patterns and a catalogue of reusable structural metrics. We are interested in stating which types of architectural attributes can be evaluated with structural metrics, and how to define them and use them in a simple way in order to make the evaluation of alternatives more systematic. Although the use of J-PRiM has been adequate for supporting the development of the Home Service Robot case study, more experimentation will be done in order to provide accurate data on the effort and benefits of using this approach in industrial case studies.

**Acknowledgements.** This work has been partially supported by the CICYT programme project TIN2004-07461-C02-01. Gemma Grau work is supported by an UPC research scholarship.

## References

1. Basili, V.R., Caldiera, G., Rombach, H.D.: "The Goal Question Metric Approach". Encyclopedia of Software Engineering, Wiley, 1994.
2. Bastos, L.R.D., Castro, J.F.B.: "Enhancing Requirements to derive Multi-Agent Architectures". In *Proceedings of WER 2004*. pp. 127-139.
3. Bengtsson, P., Bosch, J.: "Scenario-based Software Architecture Reengineering". In *Proceedings of the 5th International Conference on Software Reuse*, 1998. pp. 308-317.
4. H. de Bruin, H., van Vliet, H.: "Scenario-based Generation and Evaluation of Software Architectures". In *Proceedings of the Third International Conference on Generative and*

- Component-Based Software Engineering*, 2001. LNCS 2186, pp.128-139, 2001.
5. Bryl, V., Massacci, Mylopoulos, J., Zannone, N.: "Designing Security Requirements Models Through Planning". In *Proceedings of CAiSE 2006*. LNCS 4001, pp. 33-47.
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. ISBN 0-471-95889-7. John's Wiley & Sons Ltd, 2001.
7. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.
8. Clements, P., Kazman, R., Klein, M.: *Evaluating Software Architectures. Methods and Case Studies*. ISBN 0-01-70482-X. Addison-Wesley, 2002.
9. Dardenne, A., van Lamsweerde, A., Fickas, S.: "Goal-directed Requirements Acquisition", *Science of Computer Programming*, Volume 20, Issue 1-2 (April 1993), pp. 3-50.
10. Dobrica, L., Niemelä, E.: "A survey on software architecture analysis methods". *IEEE Transactions on Software Engineering*, Vol. 28, Issue 7, July 2002. pp: 638 – 653.
11. Fenton, N.E., Pfleeger, S.L.: *Software Metrics: A Rigorous and Practical Approach*. ISBN 0-534-95429-1. International Thomson Computer Press, 1996.
12. Franch, X., Maiden, N.: "Modeling Component Dependencies to Inform their Selection". 2<sup>nd</sup> Intl. Conference on COTS-Based Software Systems (ICCBSS), LNCS 2580, 2003.
13. Franch, X. "On the Quantitative Analysis of Agent-Oriented Models". In *Proceedings of CAiSE 2006*. LNCS 4001, pp. 495-509.
14. Franch, X., Grau, G., Quer, C.: "A Framework for the Definition of Metrics for Actor-Dependency Models". In *Proceedings of RE 2004*, pp. 348-349.
15. Genero, M., Piattini, M., Calero, C.: "A Survey of Metrics for UML Class Diagrams". In *Journal of Object Technology*, vol. 4, no. 9, November-December 2005, pp. 59-92.
16. Grau, G., Franch, X.: "ReeF: Defining a Customizable Reengineering Framework". In *Proceedings of CAiSE 2007*. LNCS 4495. pp. 485-500.
17. Grau, G., Franch, X.: "On the Adequacy of *i\** Models for Represeing and Analysing Software Architectures". To appear in *Proceedings of the First International Workshop on Requirements, Intentions and Goals in Conceptual Modelling*, RIGiM 2007 (at ER 2007).
18. Grau, G., Franch, X., Ávila, S.: "J-PRiM: A Java Tool for a Process Reengineering *i\** Methodology". In *Proceedings of RE 2006*. pp. 352-353.
19. Grau, G., Franch, X., Maiden, N.A.M.: "A Goal Based Round-Trip Method for System Development". In *Proceedings of REFSQ 2005*. pp. 71-86.
20. Grünbacher, P., Egyed, A., Medvidovic, N.: "Reconciling software requirements and architectures with intermediate models". *Software and Systems Modeling*, Vol. 3, Num. 3, August 2004. pp. 235-253.
21. The *i\** wiki at: <http://istar.rwth-aachen.de/>. Last Accessed: May 2007.
22. Kang, K.C., Kim, M., Lee, J., Kim, B.: "Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets – a case study". In *Proceedings of SPLC 2005*. LNCS 3714. pp. 45-56.
23. Kim, M., Lee, J., Kang, K.C., Hong, Y., Bang, S.: "Re-engineering Software Architecture of Home Service Robots: A Case Study". In *Proceedings of ICSE 2005*. pp. 505-513.
24. Kolp, M., Giorgini, P., Mylopoulos, J.: "Organizational Patterns for Early Requirements Analysis". In *Proceedings of CAiSE 2003*. LNCS 2681. pp. 617-632.
25. van Lamsweerde, A. "Goal-Oriented Requirements Engineering: A Guided Tour". In *Proceedings of ISRE 2001*. pp. 249-263.
26. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. ISBN: 0-13-182957-2. Prentice Hall, 1996.
27. Yu, E. *Modelling Strategic Relationships for Process Reengineering*. PhD. thesis, University of Toronto, 1995.
28. Yu, E.: "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering". 3<sup>rd</sup> IEEE Intl. Symposium on Requirements Engineering, ISRE 1997.